AFRL-IF-RS-TR-2001-293
Final Technical Report
January 2002

# ENVIRONMENT FOR REFLECTIVE AGENTS (ERA)

Crystaliz, Incorporated

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*



AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-293 has been reviewed and is approved for publication.

APPROVED: Daniel E. Daskiewich

DANIEL E. DASKIEWICH
Project Engineer

FOR THE DIRECTOR: Michael Talbert

MICHAEL TALBERT, Maj., USAF, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

| REPORT DOCUMENTATION PAGE | Form Approved OMB No. 0704-0188 |
|---|---|



**1. AGENCY USE ONLY** *(Leave blank)*

**2. REPORT DATE**
JANUARY 2002

**3. REPORT TYPE AND DATES COVERED**
Final Jul 98 - Sep 01

**4. TITLE AND SUBTITLE**
ENVIRONMENT FOR REFLECTIVE AGENTS (ERA)

**5. FUNDING NUMBERS**
C - F30602-98-C-0250
PE - 63760E
PR - AGEN
TA - T0
WU - 21

**6. AUTHOR(S)**
Sankar Virdhagriswaran

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Crystaliz, Incorporated
9 Damon Mill Square, 4D
Concord Massachusetts 01742

**8. PERFORMING ORGANIZATION REPORT NUMBER**
N/A

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Air Force Research Laboratory/IFTB
525 Brooks Road
Rome New York 13441-4505

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
AFRL-IF-RS-TR-2001-293

**11. SUPPLEMENTARY NOTES**
Air Force Research Laboratory Project Engineer: Daniel Daskiewich/IFTB/(315) 330-7731

**12a. DISTRIBUTION AVAILABILITY STATEMENT**
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*
Environment for Reflective Agents (ERA) is an asynchronous, distributed computing environment that supports the interactive design and development of distributed network applications. Using ERA's VERA editor, "casual" programmers can dynamically configure the workflow between autonomous agents, specifying which agents talk to which, and how. ERA also permits networks that are recursively constructed from existing agents and networks, in a bottom-up manner. ERA allows both direct and scripted control of intrinsically distributed, heterogeneous organizations. The activities of multiple agents are coordinated by using scripts to direct the transfer of information between them. ERA expects most substantive processing to be done by conventional agents, either people, application programs, or conventional scripts. The tools provided by ERA focus mainly on the question of who talks to whom and how. The report begins with the concepts, philosophy, and goals of ERA, then presents the initial Scheme-based ERA prototype and ERA server, which is capable of interpreting the scripting languages ERASE and DERAIL. Finally, the Java-Based ERA server is presented, together with a description of how the user can compare hierarchical, distributed agent networks interactively using the visual editor.

**14. SUBJECT TERMS**
Distributed Computing Environment, Asynchronous, Software Agents

**15. NUMBER OF PAGES**
52

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

*Table of Contents*

*Table of Figures*

# Concepts and Goals

## Organization-Scale Task Automation

ERA is a system for both direct and scripted control of distributed, heterogeneous organizations. Its intended users are "casual" system managers and programmers, as well as applications developers.

Organizations of all sizes expend considerable energy simply moving information from one agent (person or processing element) to another. Paper mail, telephone, email, web page transfers, database queries, remote procedure call, and discussion groups are just a few obvious examples. Moving and routing information are important enough that significant capital is invested in software and hardware to automate information service tasks. Some examples:

- A voice mail system is purchased to lighten the workload of receptionists
- A public web server is purchased to lighten the load on sales and support staff
- An internal file server is used to make information resources more readily available within the organization.

These are just a few examples of task automation in which a previously human-intensive activity is replaced by an automated agent, to the benefit not only of the organization but sometimes also to other agents with which it interacts (customers, suppliers, etc.).

Conventional non-distributed task automation, while not always easy, is already handled as well as one might expect using existing scripting languages, and ERA makes no attempt to compete in these problem spaces. Scripts that can run as ordinary programs on individual computers can be written in Perl, Python, VB, WebL, Rebol, or a host of other languages, and these scripts can be launched from the shell (window system) or from an application. Also in the arsenal are ASP, JSP, and PHP to improve the functionality of web servers, Javascript for web page enhancement in browsers, and Java for use in nearly every niche.

The gap ERA tries to fill has to do with organization-level tasks as opposed to agent-level tasks. ERA coordinates the activities of multiple agents by using scripts to direct the transfer of information between them. ERA expects any substantive processing to be done by conventional agents, people, application programs, or conventional scripts (as described above). The tools provided by ERA focus solely on the question of who talks to whom and how.

The thesis is that organization-level task automation is easy for the casual programmer - someone who doesn't know or want to know the details of agent connection and communication - to conceptualize, but difficult for the application programmer committed to using a conventional scripting technology to implement or conceptualize. ERA is a distributed control system from the outset, as opposed to Java, for which

serialization, RMI, and Jini are an afterthought. ERA specializes in moving information around so that it can be processed by appropriate non-ERA agents.

## Heterogeneous Organizations

ERA is concerned with controlling organizations that are intrinsically distributed. ERA is not designed for the purpose of simple single-host scripting, or even of taking a single-host program and distributing it for performance reasons. It could be used that way, but in any such situation it is likely that there is a more natural solution (e.g. a Java program is probably best distributed using RMI or Jini). We focus on the situations where distribution is inherent for one reason or another: administrative, legal, contractual, historical, security concerns, etc., and where the agents to be coordinated are of such different kinds that communication of any sort is the primary concern.

The agent to be scripted need not even be a computer. A human receiving email from a program saying that a situation resembling a server break-in has occurred, and could they kindly look into it and reply with some kind of report is effectively acting as a "subroutine" of that program, and the person who wrote the program has used the human who responds to email as they would any other agent. This view lends itself to the incremental replacement of manual operations with automated ones, since the structure of the organization as articulated by ERA doesn't change when a human's time is freed up by automation.

## Casual Programming

ERA is based on a certain theory of its customer, the casual programmer. It assumes that the casual programmer is problem-driven rather than technology-driven. The casual programmer would prefer to start out by saying in general terms what coordination needs to be done, and only afterwards to explore available technology to determine exactly how to accomplish it.

The general terms provided by ERA describe patterns of communication between components. The details of exactly what those components are, and what they say to one another, are filled in after the overall shape of the desired communication network is known.

Our model of the casual programmer's use of ERA is as follows. The programmer approaches an ERA control program, probably a GUI, and begins building a description of a desired network. There are two ways to do this: by direct manipulation of wiring diagrams, and by writing scripts. These two modes are equivalent; communication patterns are easier to visualize in graphical view, but scripts provide a more concise and familiar notation for many familiar operations. The script compiler generates wiring diagrams and performs some useful bookkeeping and helps to generate networks that move information to where it is needed.

Supporting a problem-driven attack on the programmer's problem, ERA allows the construction of network diagrams in advance of specification of component and communication details. Perhaps some initial debugging can occur at this stage with

"mock-ups" of components. The details are filled in through a process of discovery and refinement.

The discovery process is guided in part by type information -- e.g. by declarations (through the GUI or the scripting language, or by type inference over the network) that a certain kind of message or document will be transmitted over a communications link. Given a clue such as document type, a library of utilities can be consulted to find out which available components are capable of understanding that document type. If there is a mismatch between the type of the messages to be transmitted and the type of the messages understood by the recipient, a new component may be interposed (or requested) that converts from one type to the other.

Protocol may also be guided by a discovery process, although the choice of protocol will usually be determined by the choice of components being used. Change of protocol -- for example, to get through a firewall -- might be accomplished by the introduction of protocol converters.

Deployment decisions are in principle independent of component and protocol decisions, but they will often be dictated by location constraints on components -- e.g. a database server that is to be part of the network might already be "pinned" to a particular host, as a result of decisions that are not up to the ERA user/programmer.

No decisions about a system should be final. A system may be modified while it is running. Components may be replaced, rewired, or relocated as needed.

## Analysis

The analysis given here is both descriptive, interpreting the world in a certain way, and prescriptive, encouraging a certain approach to structuring certain engineering artifacts.

We take the existence of the computational world as given, and are concerned about how to understand and influence this computational world in order to bring about desired effects. This stance contrasts with that of classical engineering projects in which a new artifact is built that has its own intrinsic logic and a definite boundary. Like the Internet and the world economy, ERA is conceived of as an open system.

By "the computational world" we mean everything that might be observed or controlled by computers. The computational world is naturally thought of as a set of discrete agents connected by communication links. The agents are computers and the items that they observe and control, and the links are a combination of point-to-point cables and multi-point "ethers" such as radio (or optical or audio) frequency bands and multiple drop cables. The Internet is part of this computational world, as are local area nets, copper and cellular telephone systems, etc.

Because people interact with computers, this definition implies that people who interact with computers are agents, and part of the computational world that ERA might be prepared to deal with.

#### Agent Composition

The granularity of this abstract analysis can be varied according to need. A collection of agents or links may be composed and treated as an individual, or a single agent or link may be multiplexed and treated as if it were multiple agents or links.

For example:

- A single computer system, composed internally of multiple agents (CPU, memory system, disk, network card), might be observed on the Internet to be a single entity with no discernible internal structure.
- A data bus with many signals running in parallel is logically treated best as a single wire.
- Time-sharing makes a single CPU appear to be multiple computing elements.
- A single physical Internet link is equivalent to multiple virtual communication links when bandwidth is time-shared among multiple TCP connections.

This view is clearly important from an agent engineering perspective, since composition and multiplexing give us ways to think about forming new agents from old ones.

#### Connections

The most important aspect of an organization consisting of communicating agents is *who talks to whom.* When one agent is talking to another, or has the capability of talking to another, then the agents can be said to be *connected*, or that there is a *link* from one to the other.

Communication links come and go, so connectivity changes over time. Important organization-level questions about connectivity include the ways in which connections are formed and broken, the ease with which a connection is formed or broken, and the duration of connections once formed.

The two principal potential starting points in developing a model of the computational world are the *network model* and the *arena model.*

#### Network (actors) model

In the actors model of computation[1,2], connections are formed only through *introductions.* If an agent X is connected to agents Y and Z, then X may introduce Y to Z, allowing the

[1] Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems, Artificial Intelligence Series, MIT Press. 1986.

[2] C. Hewitt. *Viewing control structures as patterns of passing messages.* Journal of Artificial Intelligence, 8(3), 1977.

formation of a connection from Y to Z. In doing so, X may or may not retain its acquaintance with Y; if it doesn't, the transfer is a *handoff*.

**Arena model**

The model may assume an arena -- that is, an institution, facility, or space acting as a commons or meeting place -- in which agents may encounter one another. Who encounters whom follows a policy or matchmaking rule implemented by the arena. The rule may be based on random encounter, proximity, availability, template matching, addressing, etc. The analog in nature and in society is mobile agents (molecules, organisms, people), which encounter one another as they move through space, and may on occasion interact. A relationship so initiated, such as an encounter with a store clerk, may be either transitory or durable.

In developing a theory of agents suitable as a unified basis for implementations of useful services, it is necessary to decide which of these models is to be considered to be fundamental. Each has its merits and its applications, so it is worth considering each as a starting point.

ARENA MODEL

*Realism.* The arena model accurately describes many real-world organizations, in particular those in which agents do not fully control their own location or connections. Some examples:

- Solutions and suspensions in fluid media, such as molecules in cytoplasm or plankton in water. Molecules collide with one another at random; a copepod encounters food according to the whims of current;
- Mobile agents in open spaces, such as shoppers in a business district or vehicles in a road network. I do not decide which stores will be in business or which vehicles I will encounter while driving.

*Anonymity.* The arena is characterized by anonymity and interchangeability. If I am in a strange town and I need a pharmacist or locksmith, I can wander about until I find one ("discovery"). I do not need to know the name of an entity that merely satisfies a role. Parts interchangeability promotes fault tolerance, since a relationship broken due to failure of competence or communication can be replaced by another equivalent one from the arena.

*Specificity.* An arena may permit the simulation of specific communication links if it has sufficient specificity in its matchmaking capability. If I am in town and want to communicate with my pharmacist instead of just any pharmacist, I might

- Wander about until I see an agent that I can identify to be my pharmacist (by name, appearance, challenge/response, etc.),
- Sit still and wait for my pharmacist to come by,

- Communicate over a broadcast medium, assuming that messages to my pharmacist can be specific enough that everyone other than my pharmacist will ignore the messages, or
- Drop a letter in a mailbox (if the town has a postal system), assuming that the address on the envelope will determine which agent or agents will receive the message.

The last case is a special one, and describes a kind of arena that might be called a *switching network.* Well-known switching networks include the world telephone system(s), in which the telephone number entered influences the arena to direct further messages to a particular destination, and the Internet, in which the destination address field of a packet influences the arena to direct the packet to a particular intended destination.

### NETWORK MODEL

*Realism.* The world is geometric. For the most part, things that communicate actually are either next to one another or connected physically by an information conduit (broadcast media being the big exception). The notion of connection gives a way to talk about structured agents (my hand is connected to my arm) that is not natural in the arena model.

*Anonymity.* Unspecific interaction can be simulated here as just one particular kind of agent, linked to all the agents that are "in" the arena, and introducing them to one another according to whatever rule it considers appropriate. For example, a broadcast packet on a LAN can be seen as a transmission to a particular "LAN" agent, which is capable of addressing any machine that happens to tap into the LAN. This may be more than just a formal fiction forced on our thinking by a desire to use a network model, as the LAN may actually be built physically using point-to-point connections to hardware devices such as hubs and switches.

There may be multiple arenas in the organization that we wish to describe or build. For example, in a network exploiting radio frequency cells, each cell is its own arena. If an arena is just another agent, we can express concepts, such as connectivity between arenas that are difficult to think about when we think of the computational world as being, a priori, a single arena.

The value of a network model is its implicit specification of which agents do not talk to one another. By observing that there is no link from A to B, one can deduce that the behavior of B cannot be directly affected by that of A. The network makes organizations easier to understand and debug by articulating the limits on potential interactions between agents.

In ERA, the computational world is viewed as a network, not as an arena. Arenas, in the form of "discovery protocols" and "directory services", are an essential piece of the ERA design, but we have not yet focused on their development. In the future, arenas will be taken to be agents of a certain special sort, embedded in a communications network alongside other agents.

## UNINTENDED CONSEQUENCES

Unintended consequences arise from unplanned connections. If I wander around a public place, I may meet someone and strike up a conversation. Doing so is risky, and I put into place a number of complex mechanisms to reduce the risk; e.g. I can find out whether they and I have any common background or acquaintance. If they ask for money or threaten me, I can sever the connection by walking away. But the risk may pay off, because they might give me sightseeing advice or stock tips, or become a lifelong friend.

The arena metaphor describes unintended meetings more directly than the network metaphor. We do not ordinarily think of rooms, train stations, or cities as being "agents". But they serve the same kind of function as agents such as stockbrokers, dating services, bulletin boards, and advertising-laden artifacts such as newspapers.

Unintended consequences are messy, and both professional and amateur engineers try to minimize them. Even if an encounter has the potential for benefit, an automated system is very unlikely to be properly receptive to unexpected input. Because ERA was conceived as a system for the control of agent organizations through scripts developed by casual programmers, ERA has always been more interested in minimizing interactions by managing point-to-point relationships between particular agents than in maximizing interaction by arranging accidental encounters.

## DATA FLOW

When one agent says something of importance to another, we can say that information (or data) flows from one to the next. In this sense, "data flow" is a trivial observation about what happens in the world, not a particular commitment regarding the way that particular organizations are or should be programmed.

The term "data flow" calls to mind the data flow programming model, developed as a parallel computing architecture by Jack Dennis and Arvind at MIT and adapted for use in image processing and other arenas by others (Hicks, Onanian, etc.). In this model, each agent (processing element) waits until it has received sufficient information for it to be able to take some action. Taking action consists of performing a computation on its inputs and forwarding results to downstream agents.

For example, a processing element that knows how to multiply waits until it has both operands, which arrive individually. When both operands have arrived, it multiplies them, transmits the product to another processing element, and waits for further input pairs.

It is instructive to compare data flow graphs with call graphs. In a call graph, two parts of a program (subroutines) are linked if one calls the other in order to accomplish its task. If A calls B and B calls C, then there are links from A to B and from B to C. The corresponding data flow graph might have links A - B - C - B - A, but a more fine-

grained data flow graph might have links A1 - B1 - C - B2 - A2 where A1 and A2 are the parts of A that precede and follow (resp.) the call to B, and similarly for B1 and B2.

If B has any caller other than A, then the link from B2 to A2 will be dynamically mediated by a "switching network". B2 communicates with a switching network that knows how to route the data flowing out of B2 to the appropriate recipient. In this case the recipient is A2, since the data received by B1 from A1 necessarily includes "return address" information specifying A2.

## ADDRESSING

The comparison between network and arena is through the concept of addressing as implemented by a switching network.

Addressing is a fundamental, if not fully articulated, mechanism in the class of "reference-oriented" programming languages. Reference-oriented languages include any language that features objects to which one may obtain a pointer, unique identifier, or any other kind of name or address. Object-oriented and higher-order (that is, functional) programming languages are special cases, but even languages such as C and Pascal have pointers. In these languages agents deal in references to other agents, so that the choice of communication partner may be determined by a previously received reference. If an agent has the name of another agent, it is effectively linked to that other agent, by means of the switching network.

In reference-oriented languages, we don't normally think of pointers as being names. This is because the programmer never sees these names, due to an additional level of indirection. A subprogram might at some time call a particular object X, but the object's name isn't X; X is just a variable, and might mean a different object at a different time or in a different subprogram. But if you look under the hood, you will see that within a single computer's address space (RAM), objects do have definite addresses (perhaps changed systematically by a garbage collector), and in a distributed system (such as Java VM's linked by RMI) objects necessarily have names, so that references to them can be transmitted over a communications link as a definite sequence of bits.

Although we generally think of a reference-oriented system as having only a single switch or dereferencing mechanism, some additional structure can be imposed (that is, potential communications ruled out) if the space of referenced agents can be partitioned into multiple species or types. For example, if using a static type system, one can determine that some points where references are used will always possess references to cows, while other such points will always possess references to sheep, then each point may be classed as a cow use or a sheep use, and can be connected to a specialized switching network (separate address space / address bus) as appropriate.

## THE NETWORK MODEL AND THE REAL WORLD

For an agent control system to be useful, it must be capable of interacting with (controlling and being controlled by) agents that users want to deal with, including

relevant applications, frameworks, clients, servers, and protocols. The ideal system would understand all the major Internet protocols (especially HTTP and SMTP), object protocols (IIOP, RMI, FIPA), message representations (Java serialization, HTML, XML), directory and discovery protocols, programming languages, operating systems, file systems, databases, and applications. In order to control the computational world, one wants a distributed super-shell.

Mapping these notions onto our network model, we have something like the following:

| Concept | Real World Analog |
|---|---|
| Agents | **application programs**<br>**subroutines**<br>**"objects" (sensu OOP)**<br>**peripheral devices**<br>**people** |
| Links | **TCP connection**<br>**address space (RAM)**<br>**object reference** |
| Messages | **bit**<br>**number, character, string**<br>**record**<br>**packet**<br>**document**<br>**address space (RAM)**<br>**tuple space (e.g. Javaspace)** |
| Switching Agents | **WAN-IP**<br>**WAN-STMP**<br>**LAN Broadcast**<br>**JINI discovery protocol** |
| **Arena agents** | **EM broadcast (infrared, radio)**<br>**newsgroup** |

**Table 1**. Realization of ERA Concepts

## ERA Implementation

The initial implementation of ERA produced a Scheme-based, prototype ERA server, capable of hosting nodes in ERA-described agent networks. The Scheme-based ERA server interprets the scripting languages ERASE and DERAIL, in which ERA networks are expressed. The linguistic infrastructure related to ERASE and DERAIL is described below.

A Java-based ERA server was subsequently implemented, incorporating network design principles acquired during the scheme-based implementation (e.g., the use of *to do* lists to achieve *optimistic concurrency control*). A user manual for the Java-based ERA server and the companion visual editor (VERA) are included in Appendix 2.

## Overview of Capabilities

The capabilities offered by ERA can be categorized as follows:

- "Life support": facilities for daily operations. This includes a family of predefined node types; a routing infrastructure so that messages can get where they're supposed to go; two agent naming systems; handling of exceptional situations; initialization; and checkpointing.
- "Life cycle": operations for deployment, migration, debugging, and edits.
- Linguistic capabilities: the scripting languages DERAIL and ERASE.
- A Java-based ERA server, whose manual appears in Appendix 2.
- The visual editor (VERA) and event log UI, also shown in Appendix 2.

## Operations Infrastructure

ERA provides a "wiring" or "glue" infrastructure for flexible coordination of a population of agents. The infrastructure shuffles messages around, providing transmission and routing services.

### Predefined Nodes and Wiring Patterns

A basic meta-operation of ERA is to establish or disestablish a communication link going from a particular output of one agent to a particular input of another. Given a suitable population of agents, successive applications of this connection meta-operation can create arbitrary interconnect topologies.

For all other kinds of network wiring, ERA enables the creation of a variety of generally useful glue nodes. Some of these nodes are blind to message format, simply shuttling uninterrupted information around, while others are capable of combining and splitting messages in interesting ways.

The main purpose of glue nodes is to treat wiring and routing separately from the processing performed by an agent population. Many processing agents, including many non-ERA applications, can be incarnated as ERA filters, with all the non-linear (i.e. non-filter) aspects of the overall agent network handled by a small number of standard non-linear glue nodes. If nothing else, this strategy simplifies the scripting language, which can deal exclusively in filters, relegating non-filters to the network structures into which the scripts translate.

## Shuffling

The simplest of the glue nodes simply shuffle information around without manipulating it at all. Four of these are defined:

id

An identity node: It simply sends out any message that it receives, effectively splicing its incoming link to its outgoing link.

merge

A merge node has two inputs (incoming links) and one output. Any message received on either input is transmitted to the output.

copy

A copy node has one input and two outputs. Any message received on the input is transmitted to both outputs.

to-either

A to-either node has one input and two outputs. Any message received on the input is sent to one of the outputs. If an output is blocked because the node connected to it is busy, the message will be sent to the other output (assuming *it* isn't blocked as well).

to-ground

A to-ground node has one input and no output. Any message received on the input is simply discarded.

from-ground

A from-ground node has no input and one output. Nothing is ever written to the output.

spew

A spew node has one input and one output. The node emits a particular message in a steady stream. The particular message that is emitted is simply the last message received as input.

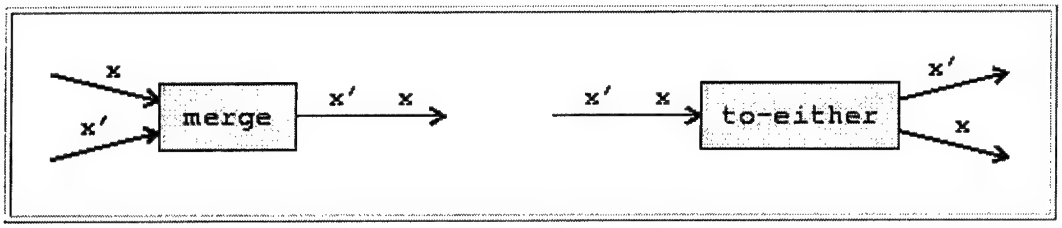



Figure 1: Glue Nodes for merging and diverging

## Coupling and decoupling

The built-in operations of coupling and decoupling form and split message pairs (couples). An operator such as addition might receive a single message containing a couple of numbers as input, and emit a single number as output. The couple can be created from messages arriving on two different connections using external wiring (a couple node). An application desiring to forward two pieces of work to different destinations might emit a single message having two parts, and then rely on external wiring (a decouple node) to split the message and route it to the two destinations.

In the Java implementation (Appendix 2), the couple concept is generalized to *tuple*, which is analogous to a vector or array.

couple

A couple node has two input connections and one output. When two messages have arrived (on the two inputs), a single message (a couple) is emitted whose two parts are the two input messages.

decouple

A decouple node has one input and two outputs. When an input message (a couple) arrives, it is split into two parts, and the parts are emitted as separate messages on the two outputs.

left

left is a filter that extracts the left part of a message (in the usual way).

right

right is a filter that extracts the right part of a message (in the usual way).

commute

commute is a filter that exchanges the left and right parts of a couple.

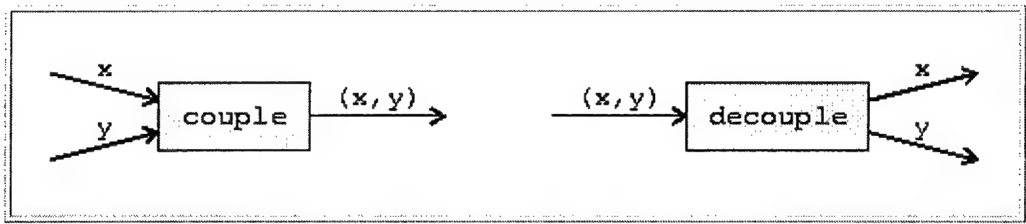



Figure 2: Glue nodes for coupling and decoupling

ERA uses the XML syntax <couple> A B </couple> for the couple of A and B, but other representations might be allowed in addition or instead, depending on the encoding scheme or protocol chosen for the particular communication link.

## Making and interpreting choices

It is often necessary to divide a single message stream out into multiple message streams by sending some messages one way and some a different way, with the choice made on the basis of some aspect of the message. This common abstraction describes both switching systems (routers, memory systems, etc.) and conditional statements in programming languages (if).
ERA provides glue nodes both for making two-way routing decisions based on a single-bit label (i.e. address), and for affixing a one-bit label so that a subsequent discrimination will be made in the desired direction.

The basic discrimination and labeling operations will generally be used in conjunction with other operators. For example, to discriminate between even and odd numbers, a filter may be used that converts numbers into the canonical labeled form required of the primitive discrimination node. Labeled messaged will then exist only ephemerally, and might even be compiled away entirely The arbitrary names "yes" and "no" are used to denote the two one-bit labels used here.

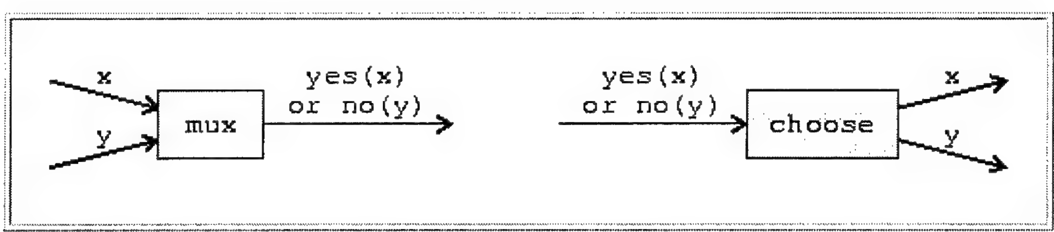



Figure 3: Glue nodes for choices

mux

Two inputs, one output. A message arriving on either input is relayed to the output, with a yes or no label affixed according to on which input it arrived.

choose

One input, two outputs. The label on an arriving message is examined. The label is removed, and the resulting message is shipped off to one output or the other depending on the classification.

yes

A filter that labels its input with a "yes" label. (Behaves the same as a mux node with grounded "no" input.)

no

A filter that labels its input with a "no" label. (Behaves the same as a mux node with grounded "yes" input.)

not

A filter that inverts the yes/no labeling of a message. Two not's in a row are an identity filter (for yes/no labeled messages).

distribute

Moves a label from the left part of a couple to the couple itself. For example, if the input is a couple of a yes-labeled message x and a message y, then the output is a yes-labeled couple of x and y. (This is an operator used in networks produced by the ERASE compiler and impossible to synthesize from the other glue operators given here.)

```
distribute((yes(x),z)) = yes(x,z)
distribute((no(y),z)) = no(y,z)
```

ERA uses the XML syntax <yes> A </yes> for a "yes"-labeled message, and <no> A </no> for a "no"-labeled message. Many other representations, such as a single bit prefixed to a message, are possible.

## Iteration

Networks of simple filters and glue nodes can be used to build both networks that iterate and networks that remember.

A network with a filter in between merge and choose nodes that are wired to one another cyclically results in an iterative process. The iteration proceeds as long as the filter emits yes-labeled messages ("yes I would like to continue iterating"), and finishes when the filter emits a no-labeled message ("no thanks, it's time to stop now").

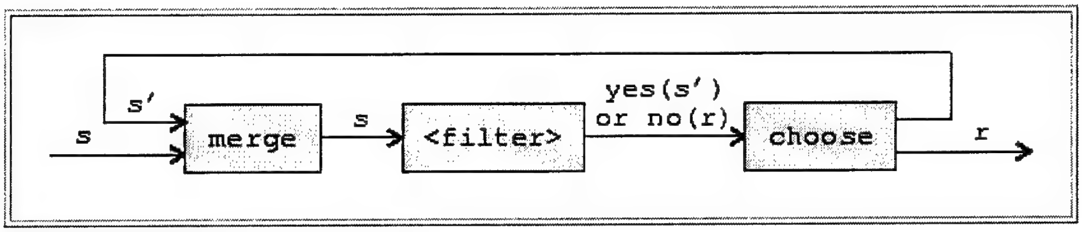



Figure 4: An iteration network

## State

Nodes corresponding to application programs and other non-ERA entities may contain their own local state. For example, a file system has file and directory contents as local state. However, it is possible to use ERA glue nodes described above to create state-bearing networks that do not require the use of non-ERA entities.

This is accomplished in a manner very similar to iteration. This time, however, the action filter combines (using a couple) its input with a "state message" stored in the network, and delivers a result that couples the value to be delivered (e.g. some information computed from the state) with a new state (perhaps the same as the old state). The left part of the couple is emitted from the cycle, while the right part feeds back to the couple node. When the filter is not active, the state resides on the arc running from the decouple node to the couple node.

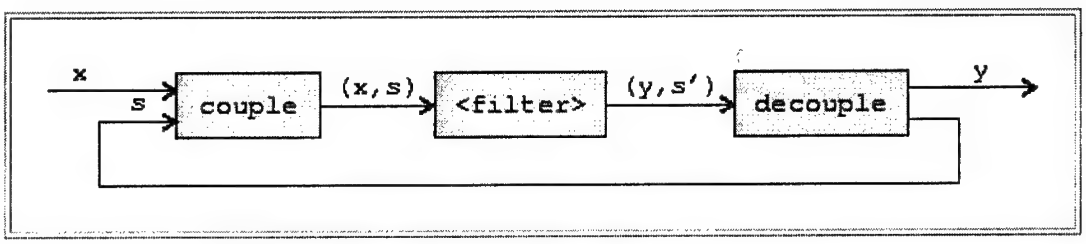



Figure 5: A stateful network

## Tasks

There is no guarantee that message pairs arriving together at a couple node are related to one another. Certain flow patterns, such as a copy connected to two arrows (filters) that connect in turn to a couple, will tend to pair outputs that are computed from the same input, but this is not guaranteed if either arrow is ill-behaved -- say, if it fails to emit an output every time, or if it emits multiple outputs for a single input.

Because it is often important to assure that outputs produced from related inputs are paired with one another and not with those produced from unrelated inputs, ERA provides a rendezvous facility that assures such proper correspondence. Before two paths

diverge, a unique "task id" [3]token is generated and stored in the memos transmitted on the two paths. Where the paths converge again, memos are paired with one another according to their task id's. Memos may arrive in any order. Excess unmatched memos represent aborted or stale tasks or runaway recursion, and are disposed of.

The following glue node types implement this task rendezvous mechanism:

fork

> One input, two outputs. Behaves the same as copy with respect to the message content, but the output messages are memos with a non-content field containing a freshly generated task id. (Actually, the task ID is pushed onto the memo's <continuation> field.)

join

> Two inputs, one output. Behaves much like couple, except that messages are matched with one another according to their task id's.

Surprisingly, in addition to dealing with faulty or ill-behaved components, the rendezvous mechanism also implements recursion. Recursion happens whenever a named agent obtains a reference to itself and re-enters before delivering an output. Suppose that the recursive re-entry occurs on one branch of a fork-join network. Values computed by the other branch will pile up the join node, waiting for their partners, which eventually arrive -- in the opposite order -- when the recursion bottoms out.

## Abstraction, Naming, and Reference

### ERA Naming Schemes

ERA natively understands two naming schemes: high-level, programmer-friendly names, and low-level switching network names. Programmer-friendly names are chosen by the ERA programmer and are scoped to a particular scripting environment, module, or library. The same programmer-assigned name may be given by different programmers (or modules, etc.) to different agents, and one agent may have multiple programmer-assigned names (different names to different programmers). Switching network names, on the other hand, are assigned and understood only by the switching network. Their purpose is to uniquely specify an agent in a context-independent manner.

In order for programmer to assign a name to an agent, the agent must first have a switching network name. At present such names are obtained in one of two ways: through the use of procedural abstraction (curry in DERAIL or lambda in ERASE), or through the basic script-structuring constructs (defderail in DERAIL, deferase in ERASE). Once a switching network name is obtained, a programmer-friendly name can

[3] The Java implementation of ERA (see Appendix 2) makes use of task ids more generally, associating a task id with every message to allow simultaneous execution of parallel tasks in the same workflow.

be assigned via a namespace operation. The defderail and deferase constructs perform both functions.

Exposing low-level agent names in this way is probably an unfortunate design choice, as it is likely to be confusing to our target audience and impossible to hide. In retrospect, it might have been better to use a more direct, low-tech, single-level approach to naming. Two-level naming is traditional in pointer-based programming languages such as Lisp and Java, however, and it does have benefits, which may in the end may make the added complexity worthwhile.

**Abstraction: Named Agents**

Low-level naming in ERA is accomplished by connecting certain named agents to a switching network (such as the Internet). The named agents must be filters -- that is, they must have a designated input and a designated output. The switching network delivers messages from various sources to the named agent's input, and from the agent's output to various destinations.

A named agent can be "used" by multiple other agents, in such a way that the "reference" to the named agent appears to be a filter embedded in a network. If A is a named agent, and there is a pipeline X - ref(A) - Y, then an output that A computes as a result of a request from X will be delivered to Y, not to some Z occurring in a different network ref(A) - Z.

How do agents acquire names? And how are those names interpreted?

Agent names in ERA are a little more sophisticated than just addresses interpretable by the switching network. Before examining the particulars of low-level ERA naming, consider the case where the switching network is the Internet. Should a name be:

- an IP address (meaning there is at most one named ERA agent at any particular host)?
- IP address plus port number (meaning that each named ERA agent at a host must have its own port and outstanding Internet "listen")?
- IP address + port + ERA-specific connection name (allowing a single ERA server process to receive messages on a single port for multiple ERA agents)?
- IP address + port + ERA-specific name + application-specific name (allowing an ERA agent to decode an address and act as a sub-switch in its own right)?

An infrastructure of network applications will generally have a hierarchy of switching networks, and it is impossible to say definitively which entities are networks and which are agents. Thus the question of where a "name" ends and where "additional identifying information" begins (the line drawn in different ways in the above scenarios) is really something that should be hedged, so that we have the flexibility to choose whatever addressing scheme we like.

ERA's agent naming mechanism assumes that any named agent may want additional information to be used for some private purpose -- information that only the agent can meaningfully interpret. (The information may be coded or encrypted, for example.) The naming mechanism always allows for the presence of such information, and if an agent doesn't need it, the information can be null and/or ignored. The above discussion implies that the information will be used for the naming of sub-agents, but it could be anything at all. For example, an application involving web browsers could use the private information to identify user-specific preferences or other context. This usage actually isn't so different from naming after all, since this arrangement causes a single agent to act as if it were multiple agents specialized to particular users.

A *ticket* for a switching network N is a datum that encapsulates the name of a named agent together with private information that is to be delivered to the named agent. Tickets are created by curry nodes and interpreted by call nodes. Agent names are assigned by curry nodes and decoded by call nodes in such a way that each agent associated with a curry node obtains a distinct name.

curry

> When given a message P, a node $curry_N(A)$ emits a ticket. The ticket will be meaningful to the switching network N as a name for agent A. The message P will become the private information in the ticket, and will be supplied to A at the point where the ticket is used.

call

> A $call_N$ node hands its input, which should be a (ticket, argument) couple, over to the switching network N. The network N decodes the ticket, and delivers both the private information and the argument to the named agent as a couple (private, argument).

ERA uses the XML syntax <ticket> ... name and private info ... </ticket> for tickets.

#### Continuations

A major difficulty with reference to named agents in ERA's distributed data flow framework is how the named agent A is to keep track of its *continuation*, i.e. how it knows to send the output of A in a network ref(A) - Y to Y and not to Z (occurring in ref(A) - Z). Clearly a description of the continuation must be transmitted to A for use when A needs to deliver its result somewhere. There are several different ways in which this might be accomplished.

Assume that A has a single node for receiving input, $A_{start}$, and a single node that transmits output, $A_{finish}$. Here are some ways in which the continuation might be communicated from $A_{start}$ to $A_{finish}$:

1. $A_{start}$ could transmit the continuation, via an independent path, directly to $A_{finish}$. When the result message catches up with the continuation, $A_{finish}$ gives both to the switching network for processing.

2. $A_{start}$ could remember the continuation, and $A_{finish}$ could route the result back to $A_{start}$ for final transmission to the next stage.
3. $A_{start}$ could transmit the continuation on to its successor(s), and the entire network for A propagates it on for use at $A_{finish}$.

Which strategy is best depends on the particular situation. The current ERA system implements strategy 3, since it requires no special wiring or internal state, and is therefore the most fault tolerant of the three.

To implement this, every message transmitted between the internal nodes of a named agent is actually a *memo*, which is a record containing multiple fields. The <content> field of a memo carries the information to be considered the primary input or output of the node receiving or emitting the memo. The <continue-with> field carries the continuation for the named agent. The continuation is ordinarily passed through an agent unaltered. When the final output of a named agent is emitted, the switching network examines the <continue-with> to decide to which referring agent the result should be directed.
Since applications of named agents can nest to arbitrary depth, the continuation is potentially a list of return points of arbitrary length. It might be nice if continuations could be passed by reference instead of value, so that this stack structure was hidden. Hiding the stack would make memos smaller and might improve both security and fault tolerance.

## ERA Servers

An operating system level process that understands the ERA network protocols and behaves as a fully functional part of the ERA world is called an "ERA server". The initial prototype server is a body of Common Lisp and Scheme code that runs under Harlequin LispWorks (although earlier versions have run in Allegro Common Lisp and in Scheme 48). The Java-based ERA server is described in Appendix 2.

An ERA server is capable of hosting nodes in ERA-described agent networks. That is, when asked to deploy a node of species x (for example, choose), it will do so. Nodes that are not connected to other nodes are implicitly connected to the local TCP/IP network (usually the Internet). Such nodes may send and/or receive messages over the Internet using the ERATIC network protocol.

An ERA server listens on a single TCP/IP port (a configuration parameter; default 6765) and responds to two kinds of messages: memos directed at particular nodes, and meta-messages directed to the server itself. The server is multithreaded and handles many nodes, and therefore uses a scheduler, with locking. An optimistic concurrency scheme is used, resulting in a robust implementation.

## Checkpoint Facility

Persistence of ERA nodes is managed in a very simple manner. Each server supports a population of nodes. A server may checkpoint that population, with their internal state, at

any time. This is accomplished in the most brutally unsophisticated manner possible. All activity is halted, and all of the nodes are written to a file, in XML form. A small amount of other server state is also written. The server may then be shut down, if desired. Regardless of whether a shutdown was clean or not, a server may be restarted at a later date based on the contents of the most recent checkpoint file.

The XML representation of a node contains

- the node's species (couple, choose, imported, etc.)
- its static parameters (the value of a constant for a constantly node, or the name of a Scheme function for an imported node)
- information about each of the node's mailboxes

Each mailbox belongs to exactly one node, so mailbox information only inside node descriptions. Mailbox information consists of

- the mailbox's handle (unique identifier)
- the contents of the mailbox, if any
- type information of any messages deposited in the mailbox
- a handle naming the mailbox's peer, if any (another mailbox)

Other information written to the checkpoint file: TCP port number, unique id prefix (in case the uid server is unreachable on restart), handles for lookup and assign agents for local namespace.

## Life Cycle Infrastructure

All directives involving network structure, initialization, finalization, debugging are handled by meta-messages transmitted to ERA servers using the ERATIC protocol. Most of these operations originate in response to a command issued in the GUI, but as they are communicable over the network, they can in principle come from anywhere.

### Initial Deployment

Node networks created by ERA are deployed either in an explicitly declared location (ERASE deploy-with or DERAIL @f), or in a default location, namely the ERA server hosting the ERASE or DERAIL compiler that compiled the script specifying the network.

#### RELOCATION

Nodes may be moved subsequent to their initial deployment. A programmer may choose to do this for load balancing reasons or to correct an erroneous initial placement decision, moving a node from where it cannot successfully perform an operation to a different server where it can.

Deletion can be thought of as a special case of relocation. Deletion is not yet implemented, but would be necessary for ERA to be complete - otherwise useless nodes would accumulate and clutter up the servers.

STATIC WIRING

A link may be established (or deleted) from an outbox on one node to an inbox on another node.

OTHER META-OPERATIONS

Here are the things that you can ask an ERA server to do.

- Examine or erase the contents of a mailbox. An inbox may be occupied if its owner is busy or not yet ready to run. An outbox may be occupied if its contents cannot be communicated to its destination, either because of communications link failure or because the target inbox is already occupied. This operation is used by the node-viewing tool.
- Provide detailed debugging information on the internal state of a node.
- Tell which ERA server is currently the enterprise-wide name server.
- "Ping": Tell whether this particular ERA server is up.
- Shut down this ERA server.
- Write a checkpoint of the server's state to a disk file.

# Linguistic Infrastructure

## The Combinator Language (DERAIL)

DERAIL is a low-level combinator language for describing certain agent networks. A DERAIL script consists of a sequence of directives whose purpose is to establish a scripting environment -- that is, the definitions of a related set of names. A defderail directive defines a name to refer to a particular agent, where the agent is described using the combinator language described below. For example:

(defderail silly (pipe left yes))

causes the name silly to refer to an agent consisting of a left node whose output connects to a yes node (i.e. silly((x,y)) = yes(x)).

Each phrase in the combinator language denotes an agent network. The networks describable in DERAIL are all arrows -- that is, they are agents with a single input and a single output. Basic phrases such as yes denote simple built-in arrows, while more complex networks may be built using various constructors.

A summary of DERAIL:

(pipe f g)

denotes the network where f's output is wired to g's input.

(pipe f g h ...)

the obvious generalization. Also, (pipe f) is f, and (pipe) is id.

(coupler f g)

denotes a fork-join network with f and g as the two branches.

(chooser f g)

denotes a choose-merge network with f and g as the two branches.

(iterate f)

denotes a cyclic merge-choose network in which one of the choose outputs is wired to one of the merge inputs.

(statefully f init)

denotes a cyclic couple-copy network in which one of the copy outputs is wired to one of the coupleinputs. (It's a bit more complex than that as there has to be a way to inject an initial value into the network (otherwise the couple node will starve); this involves attaching the init network using a merge node.)

id left right commute yes no not distribute call cocall

denote simple built-in arrows.

ignore

On receiving an input message, an ignore arrow discards that input and emits a null message.

(constantly <message>)

On receiving an input message, discards that input and emits the message denoted by <message> (for example, a number or string).

(imported <scheme>)

denotes a node that delivers an output that is the result of applying a user-supplied program written in the Scheme programming language. For example:

```
(imported (lambda (x) (* x x)))
```

designates an arrow that on receiving a number emits the number that is the square of that number.

(@f arrow place-agent)

denotes an arrow deployed at a particular place. (@f x y) behaves the same as x, but the arrow will reside at the ERA server that hosts y. y may name any agent. (Sorry about the horrible name. This is the primitive used by the ERASE deploy-with construct.)

The imported construct is quite important, as it is at present the only way for ERA to control anything outside of ERA. For example:

```
(scheme
 (define (http-fetch url)
  ;; OPEN-URL is part of the ERA implementation.
  (open-url url #f (lambda (status-code headers iport)
                (list->string
                 (let loop ()
                  (let ((c (read-char iport)))
                   (if (eof-object? c)
                     '()
                     (cons c (loop))))))))))

;; Define HTTP-FETCH to be an agent in the current script.
;; Each incoming URL is converted to the content of a document fetched
;; from the Internet via HTTP.
(defderail http-fetch (imported http-fetch))
```

## The Expression Language (ERASE)

ERASE is a scripting language that, unlike DERAIL, resembles an ordinary programming language. It has constants, variables, binary operators, calls, conditionals, and other familiar constructs.

Despite appearances, however, ERASE programs denote arrows (filters) in a manner similar to DERAIL programs. An ERASE expression is interpreted as an arrow whose input is a record containing the definitions of variables that are needed in that expression. For example, (+ x 7) is an arrow whose input is a message giving the value of the variable x and whose output is x plus seven.

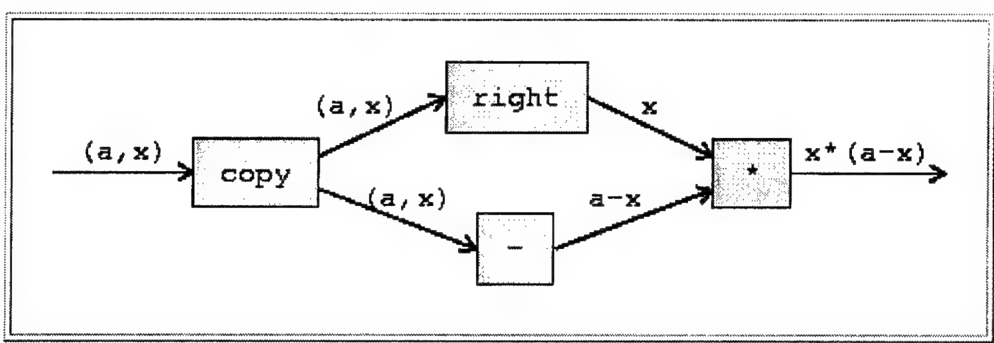



Figure 6: Network for (* x (- a x))

The record (message) holding free variable values is called an environment, although there is nothing special about it that would distinguish it from any other kind of message. If there is only one free variable, the environment is simply that variable's value. If there are N variables, the environment will have N-1 couples holding the N values.

The network for a simple ERASE expression will generally have two parts. An incoming environment feeds into a distribution network, which copies and erases different parts of the environment as needed to direct the correct values to all of the nodes that will need them. The distributed values then feed into a computation network - essentially the expression's parse tree. Intermediate values filter out to the root of the parse tree, and the final value is emitted at the root.

More advanced expressions create more complex networks, of course, including intermediate collection and redistribution networks and cycles.

# Discussion

## What We Learned

Many of the important lessons were at the implementation level:

1. Optimistic concurrency is a big help in programming multithreaded applications. Serious deadlock problems, in which locks remained locked after errors occurred, were avoided after instituting optimistic concurrency control.
2. A clean separation between the communication infrastructure and the internal operation of a node helps a lot. The design has the code implementing a node species reading and writing the node's mailboxes but not touching anything else. There is, in effect, an asynchronous process responsible for shuffling information out of outboxes and into connected inboxes.

# Appendix 1: Implementation details of ERATIC and DERAIL

## Scheduling

An ERA server maintains a set of ERA nodes. Each node has a set of actions, which can be thought of as alternative ways in which that node might make progress. An action may fail for various reasons, such as unavailability of an input or having no place to put an output. If one action fails, a different one might succeed, such as one that manipulates different sets of inputs and outputs.

Logically, each node is a separate process, repeatedly attempting to perform all of its various actions in some unspecified order. In principle, a node could be implemented as an independent operating system level process or thread. In practice, however, ERA has its own node scheduler. There are several reasons for this:

1. Most threads have nothing to do most of the time. They are simply waiting for something to change, such as the arrival of a message at an inbox, and can't make progress otherwise.
2. Most of the mechanism of a process or thread would go unused by a node, which only "waits" at a single point and therefore needs no stack. (If a node does need stack, it must manage the stack itself, since the stack is state and needs to live inside a mailbox; see below.)
3. Operating system processes/threads tend to be black boxes

For these reasons, The scheduler maintains a queue, the "ready queue", for simple round robin scheduling. Initially, all nodes known to a server are placed on the ready queue. The scheduler runs a simple loop in which it removes a node from the ready queue and runs its actions. Any node whose state changes for any reason, including a node whose action runs to completion, is put back on the ready queue.

## Communications

The scheduler runs concurrently - in alternation, actually - with the communications network. The communication network is responsible for two kinds of transaction:

1. Message transmission across static links, i.e. moving a message from one node's outbox to that outbox's "peer" inbox.
2. Message transmission through the general communication switch, as per the call and return protocols.

The two cases are distinguished by the simple and somewhat kludgey but effective convention that an outbox with no peer is considered to be connected to the communications network.

If the source and target of a message transfer are both within a single ERA server, there is no real network traffic, and communication overhead is low. Otherwise, a TCP/IP connection is established over the LAN or WAN to another ERA server. In the current

systems, a connection is set up and broken down for each message to be delivered. However, we intend to optimize this procedure when it becomes important to pay attention to performance. Connections could be kept alive with teardown after inactivity and/or on an LRU basis. There might be one connection for each destination server pair, or perhaps in high throughput situations individual TCP/IP connections devoted to peer node pairs (nodes that are statically wired to one another).

In earlier versions of the prototype, communications occurred as part of a node's actions. This had two unfortunate properties: first, the output and input were handled non-uniformly; and second, that a failure to transmit output required recomputation of the output from the inputs on subsequent attempts. After a number of bugs and confusions, we settled on the present architecture, which we believe to be more robust, easier to understand, and better for debugging. Its disadvantage is that it appears to double the number of times a message is shuffled (node to outbox to inbox to node instead of node to mailbox to node), but as this shuffling is usually just a pointer transfer.

Low-level communication in ERA follows a "push" model. If an outbox is full, then an attempt is made to contact the target, either the outbox's peer, or the `<to>` address in the memo. If the target is reachable, and has a place to store the message, then the message is transferred from the outbox to the inbox. Otherwise no operation is performed. In a better world, a stuck message, after some number of attempts or elapsed time, would result in a distress call to be transmitted to someone who could do something about it.

Communication might also operate on a "pull" model, with an empty inbox attempting to contact its peer (if it has one) to request a message. This doesn't sound very useful in the current system, which is based on forward flow, but the idea should be kept on call should it ever be needed in ERA's future.

Transmission of a memo, ERA's answer to RPC, is a sort of combination of push and pull, since the argument is pushed to a named agent, and the result is virtually pulled from it by virtue of the presence of the caller's continue-with address in the memo header.

## Mailboxes and State

Nodes have *mailboxes*, which may be either empty or occupied by a *message*. (In the current system, a message is always a memo, but this is not necessarily the case.)

Nodes aren't named, but mailboxes are. Each mailbox that needs one has a globally unique identifier called its *handle*. A handle (mailbox name) rendered in XML resembles <BOX UID='ERA-106-124'/>.

When a server starts up, it contacts a distinguished location (at Crystaliz this is a CGI script run by an HTTP server running on one of our Linux machines) to obtain a prefix to be used to generate fresh unique id. In the above example, the prefix is ERA-106-. There may be many such servers as long as they always give out nonconflicting prefixes, globally. If a server is unavailable, information on the UID generator state is read from the most recent local ERA server checkpoint.

ERA's services work best if the state of a node is visible to them. Migration and checkpointing clearly require control over a node's state, and debugging is confusing and perilous without it. A simple model of node state is essential if casual users are to be able to understand and to manipulate agent networks with safety and confidence.

## Node State Consistency

For this reason, any node desiring to play the ERA game properly must expose its state. The mechanism for doing this is to keep all state in mailboxes. The term "mailbox" is a bit misleading since mailboxes needn't be used for communication with other nodes; they may simply be used for state storage by the nodes themselves.

A second requirement for proper operation of meta-tools and accessibility to the casual programmer is *consistency*. No state should ever be observed that doesn't make sense in the context of the activities taking place. For example, an identity node -- one that simply copies its inbox to its outbox -- should never be seen in a state intermediate between the before-move state and the after-move state. Suppose an identity node initially has empty mailboxes, and a message M arrives at its inbox. An outside agent must observe either M in the inbox and nothing in the outbox, or nothing in the inbox and M in the outbox. States where M is "in transit" or "being copied", such as empty mailboxes (followed at some point by M appearing in the outbox) or both mailboxes containing M (followed by M being erased from the inbox) must never be observed. The states may (and must) exist ephemerally, but they must never be seen.

Protection against observation of intermediate states is important not only for the sake of meta-tools, but also because the communications infrastructure itself both observes and modifies mailboxes.

To achieve consistency, all activity of a node must appear to occur atomically. Ordinarily in a concurrent system atomicity is achieved using locks. As a node runs, it either locks each mailbox for exclusive access as it observes or modifies it, or (as with monitors / Java synchronized) it lock its entire state (all of its mailboxes) for exclusive access at the outset. When all operations for the current transaction are complete it releases all locks.

We initially tried this kind of concurrency control in the ERA node implementation. While debugging various parts of the system - the node scheduler, the communications network, and the logic of the various species of node - we often ran into problems of locks being left locked for too long, usually due either to bugs in nodes or bugs in Scheme 48's lock handling. These problems were alleviated by the employment of *to do* lists, which enforce *optimistic concurrency control.*

The purpose of a to do list is to reduce the period of time during which a set of resources (mailboxes) is locked. The operation that takes place while the resources are locked will definitely either succeed of fail in a very short period of time.

Here is how it works. A program that doesn't care about presenting consistent states to observers simply reads and writes its state variables with abandon. To ensure consistency,

it instead reads and writes variables using a transaction supervisor -- substitute versions of read and write that somehow prevent exposure of inconsistent states. The transaction supervisor used in a database manager progressively locks each variable that is read or written so that any attempt by another agent to read or write locked variables will block until the lock is released (when the transaction commits).

The ERA transaction supervisor, on the other hand, instead creates a to do list. The to do list holds two kinds of things:

1. Values that the agent proposes to write to its state variables. This makes sure that the new values won't be seen until the transaction finally commits, guaranteeing consistent views even while the transaction is in progress.
2. Items of the form "check to make sure that variable X still has value V". These items ensure that a consistent set of variable values will be written when the transaction commits.

To read a variable under this scheme, the agent first looks in the to do list for a previously written or read value, and uses one if it's there. If not, the true state is observed and recorded as a check-it item in the to do list. To write a variable, the agent overwrites a previously written value in the to do list, if there is one, and otherwise adds a new item to the to do list.

When the modifications are complete, the transaction attempt to commit itself by a simple atomic action. First, all check-it items on the to do list are checked. If any variables have been modified from without, the to do list is discarded and the activity is restarted from the beginning. This is safe because no state variables have been written and no locks have been taken out. Second, having passed this test, all variables to be written are given their new values.

Atomicity of commit can be enforced by taking an exclusive lock on the entire computer, if necessary, since the commit will complete in very short order. In practice one can of course do better than that.

*Detail*: An agent performing an optimistic transaction may observe inconsistent states. For example, suppose that the sum A + B must be constant across transactions. If the agent observes A, then a second agent consistently changes A and B together, and then the agent observes B, then the observed B may not be consistent with the previously observed A. However, the transaction observing the inconsistent state can never commit, since it will have a failing not-modified check for A on its to do list. As long as the agent will not do any damage as a result of observing the inconsistent state - and it cannot if it is following the regime - the transaction will abort and will be retried, and all will eventually be well.

# Appendix 2: ERA User Manual

We include here the user manual is for the Java-based ERA server and visual editor (VERA).

ERA is an asynchronous, distributed computing environment that supports the interactive design and development of distributed network applications. Using ERA's VERA editor, "casual" programmers can dynamically configure the workflow between autonomous agents, specifying which agents talk to which, and how. Moreover, ERA enables the distributed application designer to define and invoke hierarchical agent networks that are recursively constructed from existing agents and networks, in a bottom-up manner.

## Installing ERA

ERA relies on HTTP communication between *ERA servers* running on different hosts or virtual machines, so an HTTP server such as Apache or IIS is required. ERA also requires a Java servlet engine, such as Jakarta Tomcat. In addition to ERA's Java classes, the following jar files are required, and they must be included in the classpath of the computer system or Java servlet (Jakarta Tomcat additionally requires jasper.jar, servlet.jar, webserver.jar, and xml.jar):

- Java 2: tools.jar
- Rhino: js.jar
- J2EE: j2ee.jar, jhall.jar, ejb10deployment.jar
- Apache XML: xerces.jar, xalan.jar, xalanservlet.jar, stylebook-1.0-b3_xalan-2.jar
- Sun XML: sunxmlparser.jar
- Clark: xp.jar, xt.jar

A number of Java environment variables are also required by ERA, mainly for event log settings. If Jakarta Tomcat is used for the servlet engine, these variables can be defined as parameters in the web.xml file.

- EventLogDataSource: event log ODBC data source name
- EventLogDataSourceLoginID: event log ODBC data source login ID
- EventLogDataSourcePassword: event log ODBC data source login password
- EventLogTable: event log database table name
- EventLogUpdateIntervalInSeconds: event log database update frequency
- EventLogPersistenceIntervalInDays: longevity of event log database entries
- EventLogPriority: Event log Java process priority
- Repository: name of the repository directory
- ErrorReports: name of the error reports directory

## Era Nodes

ERA agents are usually interconnected in a network, so we refer to them as ***Era nodes***. A node is either a simple ***primitive node*** provided by ERA, or a ***composite node***, which may be arbitrarily complex. The ERA programmer designs composite nodes as hierarchical networks of interior nodes, which may be primitive or composite nodes. ERA provides a set of primitive node types, from which composite nodes can be recursively constructed.

Nodes communicate with each other asynchronously, through the transferal of messages between ERA node ***mailboxes***. Mailboxes are buffered ports that queue the node's incoming and outgoing messages, and store internal state parameters. A node's ***inboxes*** and ***outboxes*** store input and output data, respectively. A peer-to-peer connection is established between pairs of inboxes and outboxes and collectively all the connections dictate the patterns of dataflow in a network of nodes. Internal node state is stored in ***internal boxes***, which do not have peer connections.

A ***node definition*** is an abstraction that specifies the behavior of a type of node, separate from each instantiation and invocation of the node. ERA provides "built-in" definitions for all the primitive nodes, and the ERA programmer may create composite node definitions that recursively combine "uses" of existing node definitions, to form a hierarchical network. A composite node definition declares the component "node uses" (references to existing definitions) and specifies the mailbox connections to be established. Each primitive node definition lists the required mailboxes of the primitive node, i.e., the names of mailboxes that must be present in any use of the given primitive node. ***Mailbox types*** are optionally specified in all node definitions, stating the type or supertype of message that a mailbox can store.

A composite node use is "***instantiated***" (as a node "instance") when ERA deploys the network represented by a node definition. Because composite nodes are hierarchical, the instantiation of a composite node may lead to the recursive instantiation of internal composite and primitive nodes. Mailbox type declarations are checked for consistency when a node is deployed. If any required mailboxes are missing, or mailbox peer types are inconsistent, ERA issues an Error message stating the violations.

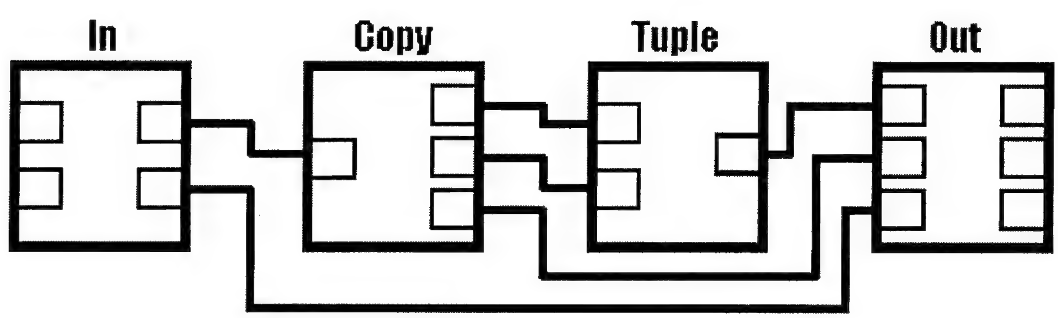



Figure 7: A Gizmo composite node

Every composite node has a pair of primitive nodes dedicated to receiving input messages and posting output messages. In Figure 7, for example, a ***Gizmo*** composite node has an ***In*** node and an ***Out*** node. These nodes enforce synchronized transferal of messages from their inboxes to their outboxes, only doing so when all of the inboxes have messages available, and all of the outboxes are free to accept them (that is, when their message queues are not full). In contrast, ***AnyIn*** and ***AnyOut*** nodes transfer messages from inboxes to outboxes whenever possible, in an asynchronous manner. A composite node can use either type of input or output node.

## Agent Execution Environment

ERA nodes are instantiated and invoked within a "***node manager***". This is an execution environment that is responsible for handling meta-level requests to deploy, destroy, connect, and disconnect nodes. The node manager is also responsible for scheduling the actions associated with the deployed nodes, and for passing normal messages between connected nodes. HTTP communication with a node manager takes place by way of an ERA server. Several node managers may be associated with a single ERA server, while only a single ERA server can live on a given host. Meta-messages containing node manager commands are sent to the ERA server in the form of HTTP POST requests. The VERA editor can send a RunDefinition command to a node manager, instantiating an instance of a named composite node type, with input messages supplied by the user.

Once a node has been instantiated, it may be "invoked" any number of times, typically in response to the dataflow of incoming messages. Primitive nodes are invoked as atomic actions, in which all of the relevant mailboxes are updated synchronously, according to the behavior associated with the primitive node type. Each primitive node has an ***action rule*** that fires whenever the state of the node's inboxes or outboxes changes. If all of the conditions specified in the action rule are met, then the atomic action is executed. The action rule for an ***In*** node, for example, specifies that all inboxes must be full and all outboxes must be empty, in order to execute the atomic transfer of messages from the inboxes to the corresponding outboxes.

## ERA Node Namespace

Every instantiated ERA node receives a unique name by which the node is identified in event log entries. The ERA node namespace is hierarchical, and each name has the form of a URL. A node name URL starts with the host of the ERA servlet, followed by the node manager index, and a user-supplied identifier for the node instance. Interior node names are inherited from the "node use" IDs of the instantiated node definitions. For example, the node name

**magenta/0/mynode/tuple1**

refers to the an interior node "tuple1" of a composite node instantiated as "mynode", in the 0th node manager on host "magenta". Multiply-deployed nodes of the same name are distinguished by parenthesized indices, e.g., mynode, mynode(2), mynode(3), etc.. If the user supplies no name, the node is named "rundef" by default. The VERA editor does not currently accompany RunDefinition commands with node names, so the deployed nodes receive the name "rundef".

## Message Types

All communication between ERA node managers, ERA nodes, and the VERA editor is accomplished by passing XML-writable Message objects, or their XML-serialized representation. Messages communicated between mailbox peers are passed in the form of Message objects when the peers reside in the same node manager. Messages transferred between nodes living in different node managers are passed in XML-serialized form.

The following message types are defined in ERA. They include primitive data types (Boolean, Double, String, etc.), compound data types (Tuple, Struct), ERA system data types (Error, ErrorReport, Event), and XML-writeable representations of ERA nodes and mailboxes (Node, NodeDef, Mailbox, MessageQueue).

- **Boolean:** Message representation of a true/false Boolean value.
- **Double:** Message representation of a double-precision real value.
- **Error:** Brief description of a deployment or runtime error.
- **ErrorReport:** Detailed description of an error, including the nodes and mailbox contents relevant to the task in which the error occurred.
- **Event:** ERA system event (e.g., mailbox transferral, action rule completion, etc.) to be written to the event log.
- **Generic:** Wrapper for a user-defined message type.
- **InstanceHandle:** Pointer to a Java instance, produced by an active node and passed as an argument to other active nodes.
- **Integer:** Message representation of an integer value.
- **Labeled:** A Boolean wrapper for an ERA message, containing a "yes" or "no" tag to indicate the result of a meta-request.
- **Mailbox:** Message representation of an ERA mailbox, containing its name, type, peer, and list of MessageQueues, one per active task.
- **Memo:** A meta-message to be processed by a node manager. Memo types mainly correspond to types of node manager requests (e.g., DEPLOY, CONNECT, etc.). All messages sent to a node manager, from a VERA editor or another node manager, are wrapped in Memos. Messages sent to Passive nodes via Java RMI, as well as all messages stored in ERA node mailboxes, are wrapped in MAILBOXMESSAGE Memos.
- **MessageQueue:** Mailbox buffer of messages associated with a specific task.
- **Node:** Message representation of an ERA node to be serialized (for an error report or for redeployment), or to describe a "node use" in a node definition. The node name (id), type and mailboxes are included.
- **NodeDef:** Message representation of an ERA node definition, produced by the VERA editor and interpreted by the node manager for node deployment.
- **Nothing:** Message representation of a "null" value or "void" response.
- **String:** Message representation of a string value.
- **Struct:** A set of name-value pairs, each associating a string-valued name with an ERA message.
- **Tuple:** A list of ERA messages.

## Primitive Nodes

There are several types of primitive nodes, with characteristic behaviors that are encoded in their action rules. The primitive node types can be divided into three classes: ***dataflow*** nodes, which simply define the flow of data through the network; ***data*** nodes, which relate to structured data; and ***meta*** nodes, which invoke or communicate with processes outside of ERA, or serve as the I/O interface for a composite node.

**Dataflow nodes**

- **Copy:** Any message arriving at a single inbox named "in" is copied to all of the outboxes, as soon as they are all available for writing.
- **Merge:** Any message arriving at any inbox is passed through to the single outbox named "out", as soon as it is available for writing.
- **AlternateIn:** This node is similar to a merge node, except that the inboxes are read in strict sequence, from one invocation to the next.
- **AlternateOut:** This node is similar to a copy node, except that the outboxes are written to in strict sequence, from one invocation to the next.
- **Mux:** This node waits for distinct pairs of messages to arrive at its two inboxes "in" and "control". The "control" box contains a Tuple message naming a subset of the node's outboxes. The message found in the "in" inbox is copied to all of the named outboxes, as soon as they are all available for writing.

**Data nodes**

- **Tuple:** This node waits for all of its inboxes to receive messages, wraps the messages in a Tuple, and writes the Tuple to the single outbox named "out", as soon as it is available for writing. A Tuple node can optionally contain a set of internal boxes, whose nonvarying messages are prepended to any outgoing Tuple.
- **Detuple:** Each Tuple message arriving at a single inbox named "in" is "detupled" to obtain its component messages, which are written to the corresponding outboxes (according to Tuple indices), as soon as they are all available for writing.
- **Struct:** This node waits for all of its inboxes to receive messages, wraps the messages in a Struct of name-value pairs, and writes the Struct to the single outbox named "out", as soon as it is available for writing. A Struct node can optionally contain a set of internal boxes, whose names and nonvarying message values are added to the name-value pairs of any outgoing Struct.
- **Destruct:** Each Struct message arriving at a single inbox named "in" is "destructed" to obtain its component name-value pairs, and the named values (messages) are written to the indicated outboxes, as soon as they are all available for writing.
- **ParallelStruct:** This node builds multiple Structs in parallel. A pair of messages arriving at two inboxes named "id" and "keys" specifies each new Struct. The "id" box gives a unique ID for the Struct, so that multiple Structs can be referenced separately, and the "keys" box gives the field names of the new Struct. The other inboxes receive Struct messages containing sets of name-value pairs, to add to one of the Structs being built. Each input Struct's "id" field indicates which Struct-in-progress should receive the name-value pairs. Once a Struct receives all of its required fields, it is written to the single outbox named "out", as soon as it is available for writing.
- **SequentialDetuple:** Each Tuple message arriving at a single inbox named "in" is "detupled", and its component messages are sequentially written to the single outbox named "out", whenever it is available for writing.

**Meta nodes**

- **In:** This node waits for all of its inboxes to receive messages and transfers them to the outboxes with corresponding names, as soon as they are all available for writing.
- **Out:** This node is functionally equivalent to the In node. In and Out nodes are treated as separate node types because of their distinct roles in a composite node, where they handle input and output messages, respectively.
- **AnyIn:** This node waits for any inbox to receive a message and transfers the message to the outbox of the corresponding name, as soon as it is available for writing.
- **AnyOut:** This node is functionally equivalent to the AnyIn node. AnyIn and AnyOut nodes are treated as separate node types because of their distinct roles in a composite node, where they handle input and output messages, respectively.
- **Synchronous:** This node invokes a Java method for every set of arguments received in its inboxes. The result of the method is passed to the single outbox named "out". A single Java class instance is created for the Synchronous node when it is deployed. The Java class and method names are specified in internal boxes "class" and "method", and the inbox types determine the method signature. An optional internal box "properties" can supply Java instance properties to set, using the bean convention that a "setFoo" method exists for every property name "foo". ERA converts ERA message properties and arguments to and from Java Objects and types whenever necessary. Tuple messages map to Java Object arrays, and Struct messages map to Java Hashtables. Alternatively, an inbox or outbox can be made to store Java object references by declaring the mailbox to be of type ***InstanceHandle***.
- **Invoke:** This node is similar to the Synchronous node, except the method is provided dynamically by way of an inbox called "method", and the arguments arrive in the form of a Tuple, in an inbox named "args". The method signature is determined dynamically by the method name and Tuple's message types.
- **Eval:** This node evaluates a fixed JavaScript expression for every set of inputs received in its inboxes. Incoming messages are plugged into the JavaScript expression, as variables named after the inboxes. The outboxes receive the resulting values of namesake variables in the evaluated expression. The single exception to this rule is that an outbox named "out" receives the overall result of the expression, if such an outbox is present. ERA message argument conversion to and from JavaScript is similar to that described for Synchronous nodes.
- **Decide:** This node is a variation of the Eval node, in that it treats the inbox names as JavaScript variables. A Decide node associates a JavaScript predicate expression with each outbox, and the input is directed to the first outbox whose expression evaluates to *true*. The chosen outbox receives a Struct message associating the inbox names with their contents.
- **Passive:** A Passive node serves as a remote interface to an external Java program. The Java program can send and receive ERA messages through the use of PassiveNodeProxyInterface methods *put* and *get*. The *put* method takes a MAILBOXMESSAGE Memo argument containing a Tuple, which the Passive node stores in its "ins" internal box until it can distribute the detupled messages to

its outboxes. The return argument is a Labeled message, whose yes/no value indicates success or failure. A successful put returns a yes-Labeled message containing a new task ID, in the form of a String message. The get method takes the task ID as an argument and fetches the contents of the "outs" internal box, which is a Tuple constructed from the Passive node's inboxes, or *null*, if not all of the inboxes have yet received input. A passive node can optionally receive LDAP identifiers (C, O, OU, CN), to enable Java programs to locate a particular passive node by name.

- **Mailer:** The Mailer node sends mail to an SMTP server. Its inboxes are "in", "from", "to", "subject", "mime" (e.g., text/plain or text/html), and "smtpHost", all of which accept String messages. The "to" inbox can either take a single recipient, or a Tuple of recipients.

## Composite Nodes

A composite node's behavior is essentially defined by the action rules of its constituent primitive nodes, and by the mailbox connections set up between the mailbox peers. Composite nodes can be composed of primitive and composite internal nodes, so a composite node can represent an arbitrarily deep hierararchy of nested ERA node networks. The instantiation of a composite node can be viewed as a recursive macro expansion of composite node definitions, which bottoms out with the instantiation of the consituent primitive nodes.

Figure 8 illustrates the sequence of network states that a *Gizmo* composite node undergoes after receiving a pair of input messages. Each successive state results from the successful run of an action rule, or the transferral of messages between mailbox peers. After the *In* node messages are synchronously passed from its inboxes (1) to its outboxes (2), the two messages are transferred to inboxes of the *Copy* and *Out* nodes (3). The *Out* node's action rule cannot succeed until all three of its inboxes are full. The *Copy* node's action produces multiple copies of the incoming message (4), which are transferred to mailbox peers in the *Tuple* and *Copy* nodes (5). The *Tuple* node's action wraps its input messages in a *Tuple* message, which is written to the outbox (6), and passed on to the mailbox peer in the *Out* node (7). All of the *Out* node's inboxes are now full, so its action rule finally succeeds, synchronously transferring the three messages to the corresponding outboxes. The three outbox messages represent the output of the composite node.

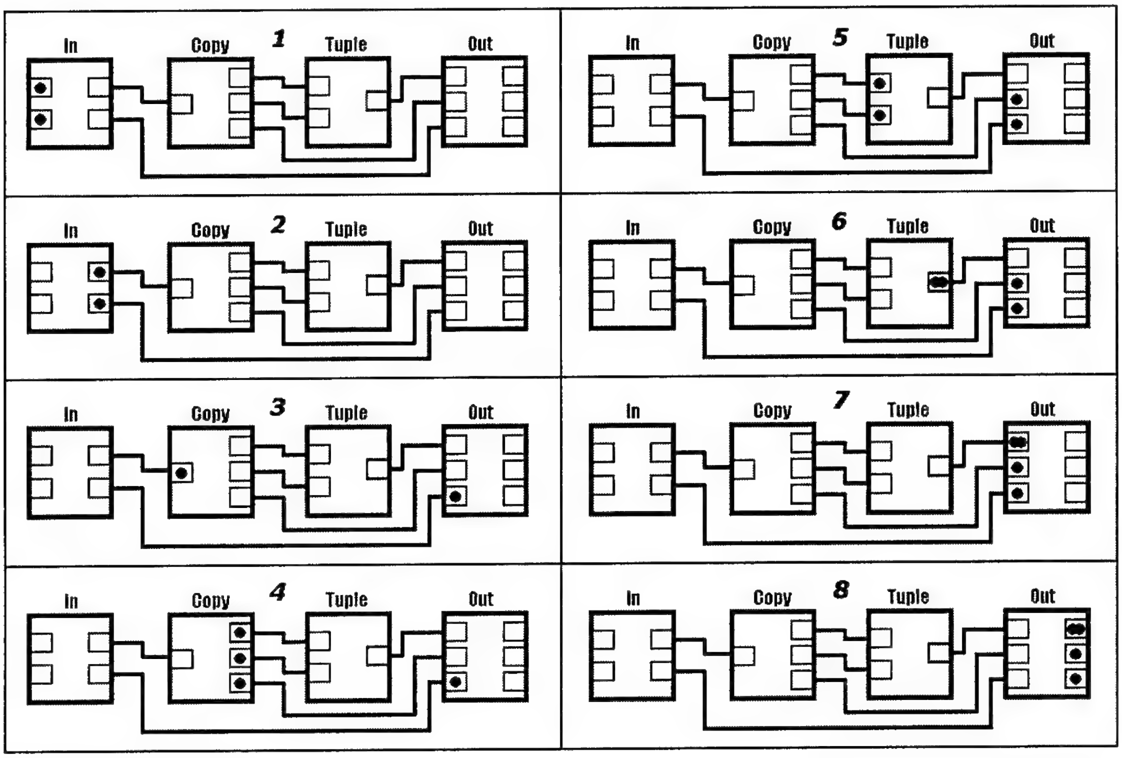



Figure 8: Sequence of states in Gizmo composite node

Most meta-nodes perform processing tasks beyond the simple transferral and collating of messages. These nodes generally execute Java methods or JavaScript evaluations, operating on message Tuples stored in internal boxes named ***ins*** and ***outs***, by convention. The action rules transfer messages from the inboxes to the ***ins*** box, where they are wrapped as a message Tuple, and from the ***outs*** box to the outboxes, which receive unwrapped Tuples of messages. For *Synchronous* and *Invoke* nodes, a Java thread executes a Java class method on any Tuple of arguments arriving in the ***ins*** box, passing the result to the ***outs*** box. JavaScript nodes of type *Eval* and *Decide* similarly operate on Tuples of arguments found in the ***ins*** box, placing the output in the ***outs*** box. A *Passive* node operates in an "inside-out" manner, receiving an input Tuple in its ***outs*** box via RMI, and outputting a Tuple to its ***ins*** box, where it can accessed via RMI.

Figure 9 shows the dataflow that takes place in these types of meta-nodes when they are interconnected in a composite node. A *Passive* node serves as the I/O entry point for the composite, enabling an external Java process to pass in a Tuple of messages as input. The PassiveNodeProxyInterface ***put*** () method places the message Tuple in internal box ***outs***, and the ***get***() method eventually obtains the result of the composite invocation, once it arrives in the ***ins*** box. In this example, the input messages are passed to an *Eval* node, where a JavaScript expression is evaluated to obtain the input (method and arguments) for an *Invoke* node. The latter node's result is passed back to the *Passive* node, to be fetched remotely via RMI

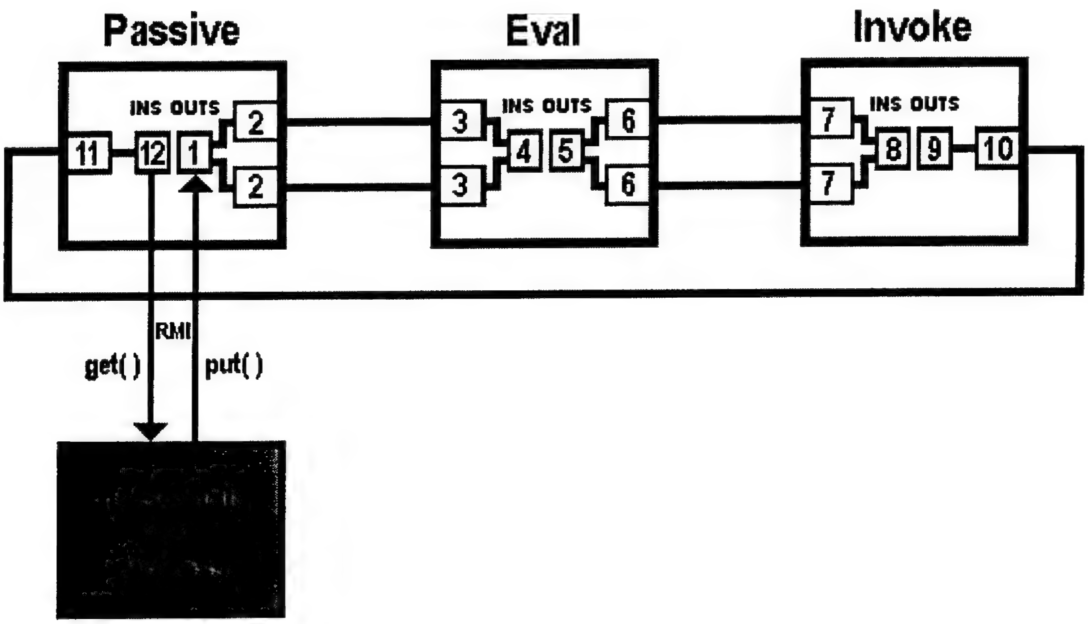



Figure 9: Data flow in Passive, Eval and Invoke nodes

## Tasks

ERA executes action rules and transfers messages in the context of distinct *tasks*. Action rules execute with respect to specific tasks, whose associated messages are kept in separate message queues in the node mailboxes. The separation of messages related to different tasks makes possible the parallel execution of multiple dataflows that share the same ERA nodes and mailboxes, without interfering with each other.

Messages flowing through an ERA network are wrapped in Memos, which are tagged with task IDs. Task IDs are generated by the PassiveNodeProxyInterface ***put***() method. The task ID returned by ***put***() is passed as an argument to the ***get***() method, to ensure that the ***get***() result will correspond to the correct task. When a RunDefinition command executes, the ERA servlet communicates with the instantiated composite node by way of a *Passive* node that the servlet instantiates along with the composite node instance.

## VERA Editor

VERA (Visual-ERA) is a browser-based visual editor for designing node networks. VERA is primarily an editor, but in future versions, it will serve as a runtime debugging tool, monitor, and administrative tool. A *Run* operation is currently provided, to instantiate a composite node of any given type, passing user-supplied input messages to the node's inboxes.

VERA allows you to create and modify one node definition per window. You can drag and drop a node from a file system-like outline view of a "Node Repository" to add a new use of that node in the current definition. You can then make connections between nodes by dragging between the in-boxes and out-boxes of each node, starting at either end. If the node

or box you want to connect to has not yet been created, you can drop the end of a connection anywhere and pick it up later to finish the connection.

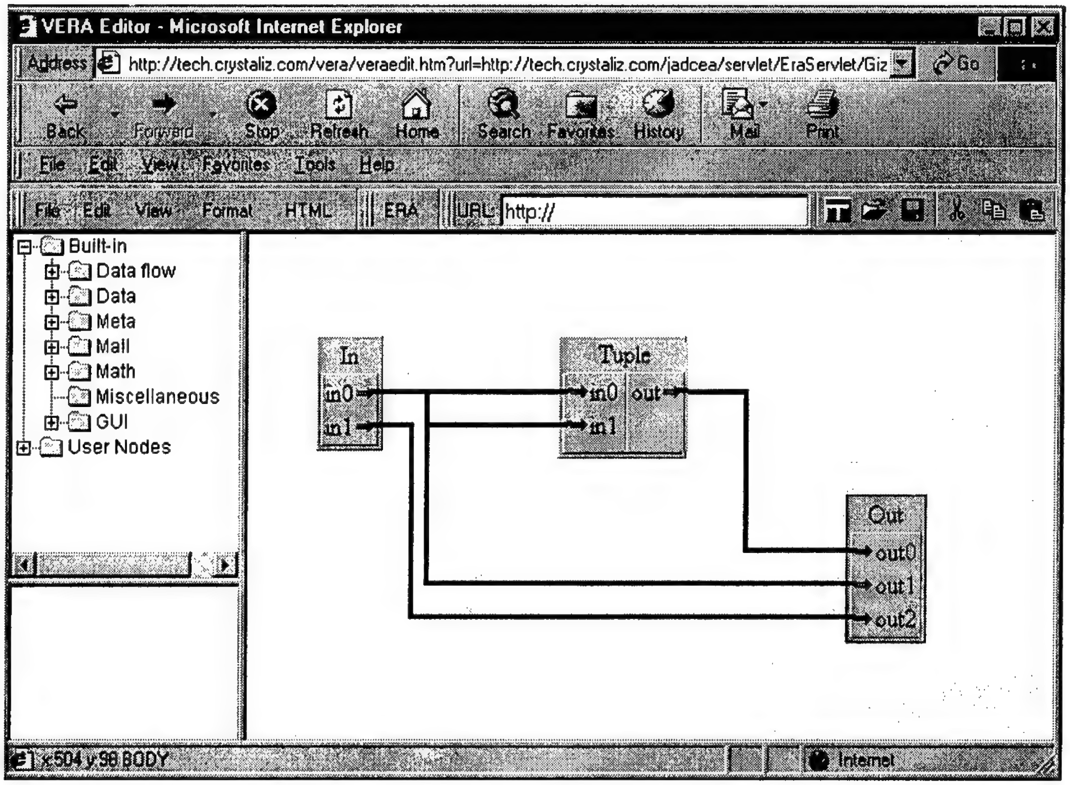



Figure 10: A VERA editing session for Gizmo composite node

A composite node definition may consist of any number of nodes within the scrollable view, but it is a good programming practice to keep the size of a definition down to that which is viewable without scrolling. Figure 10 shows an editing session for the *Gizmo* composite node example, which first appears in Figure 8. Note that VERA displays *Copy* and *Merge* nodes implicitly, as forks and merges in the connections. The forks and merges are expressed as Copy and Merge "node uses" when the composite node definition is written to the repository.

As mentioned earlier, a composite node definition has two special primitive nodes of type ***In*** and type ***Out***. (or optionally, ***AnyIn*** and ***AnyOut***), whose mailboxes serve as the input and output boxes of the composite node. You may add, delete, or rename the available boxes of these nodes, and connect their boxes up to the boxes of internal nodes.

As shown in Figure 11, you may save a definition to the ERA repository (assuming you have write access) using the Save-As dialog. The editor will submit the XML encoding of the definition to the ERA server using an HTTP POST. The URI of this saved definition is then made available in the repository outline of node definitions, so that it may be used in other definitions. Single-clicking on a node in the repository outline will show some information about the node, enough to decide whether to use it, whereas double-clicking on the node will pop up a new editor window for viewing and (if permitted) allow you to modify the node definition.

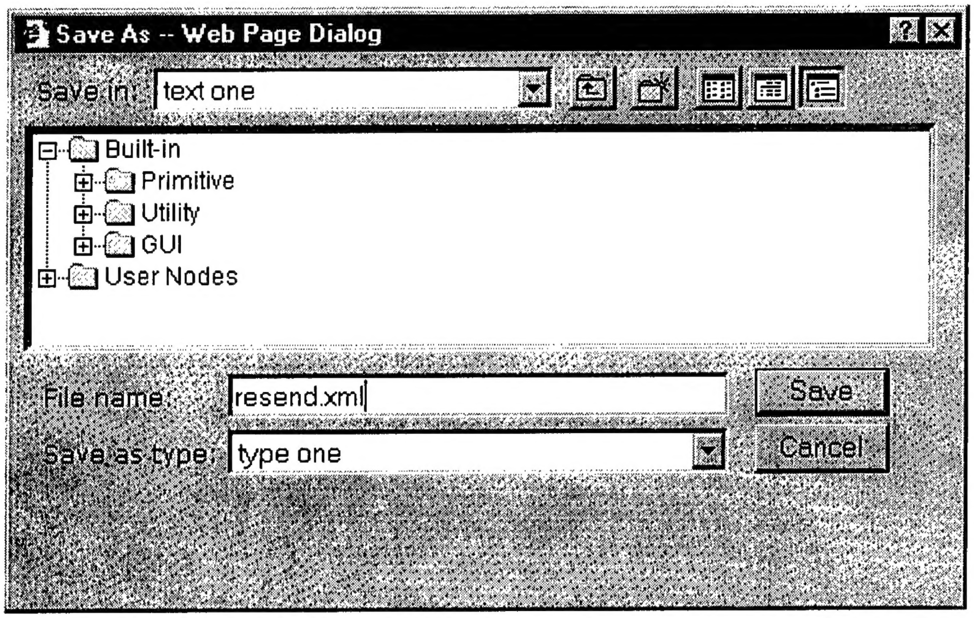



Figure 11: Save-as dialog in web page

## Error Handling and Debugging

ERA has basic facilities for catching and handling node deployment errors, as well as run-time errors. Node deployment error handling includes type checking of mailbox peers, checking for manditory mailboxes in primitive "node uses", and failing to find a Java class or method specified in a *Synchronous* or *Invoke* node. XML Parsing errors are also caught when loading a node definition from the repository. Ia node deployment error occurs during a RunDefinition operation executed from the VERA editor, an *Error* message is returned, containing a brief description of the error.

Run-time errors in ERA include Java exceptions (for *Synchronous* and *Invoke* nodes), JavaScript parsing or evaluation errors (for *Eval* and *Decide* nodes), and Message type mismatches (e.g., a Detuple node inputting a message other than a Tuple message). When a run-time error is caught by ERA, the task involved in the error is suspended, and an ErrorReport message is created and stored in the designated ErrorReports directory. The ErrorReport message contains a description of the error, along with a serialized description of each node that was previously or currently involved in the task. If a Java exception is caught by ERA, the Java stack trace is also included in the error report.

To further facilitate the debugging of composite nodes, ERA provides an ***Event Viewer,*** with which to examine past events or view events as they occur in an active network. Types of events logged by ERA include the following:

- **NEWTASK:** Creation of a new task ID, typically via a Passive node ***get***().
- **DEPLOY:** Deployment of a composite or primitive node.
- **ACTION:** Successful completion of a primitive node's action rule.
- **READ:** Destructive read of an inbox message.
- **WRITE:** Outbox message write.

As shown in Figure 12, the Event Viewer contains three panes: the *Node Tree Pane,* which displays the node hierarchy of currently instantiated nodes; the *Node Contents Pane*, showing all the nonempty node mailboxes that are descendents of the currently selected tree node; and the *Events Pane*, displaying past or current events.

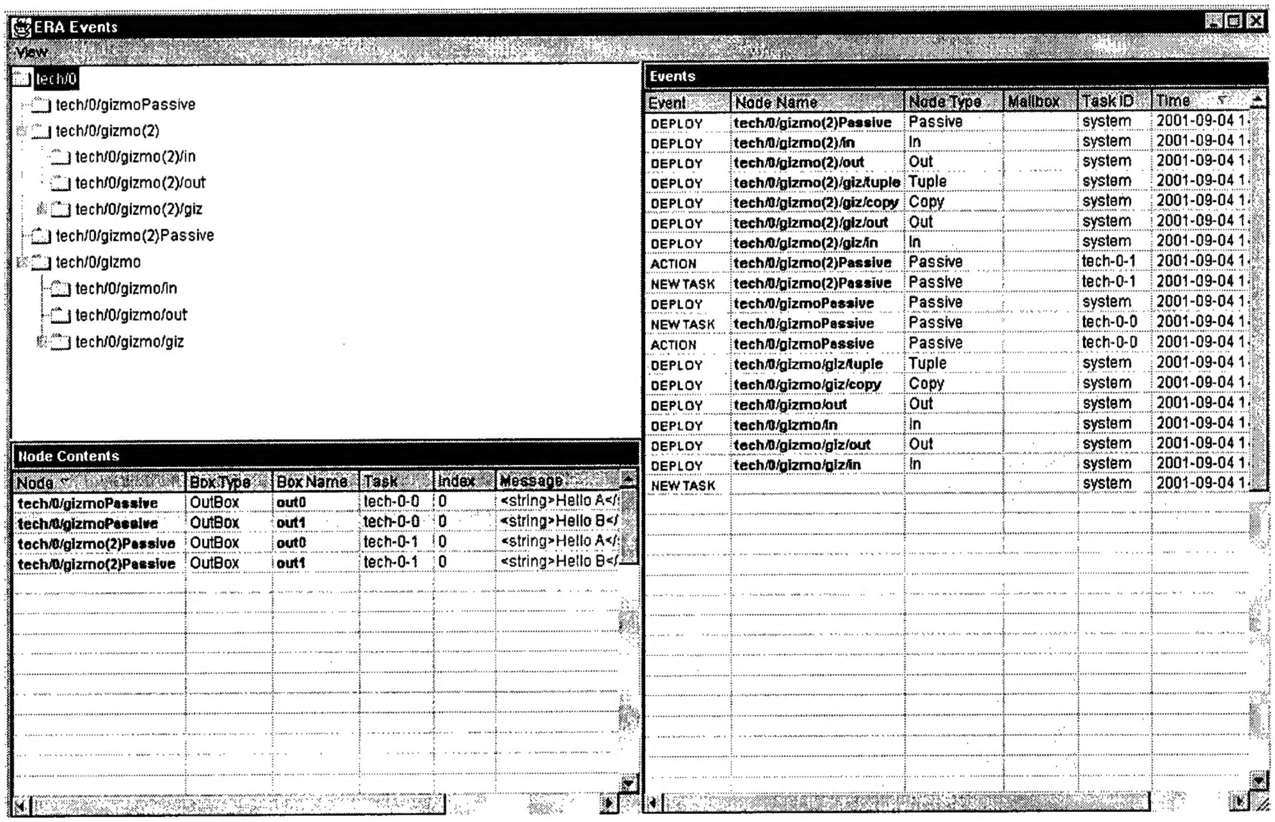
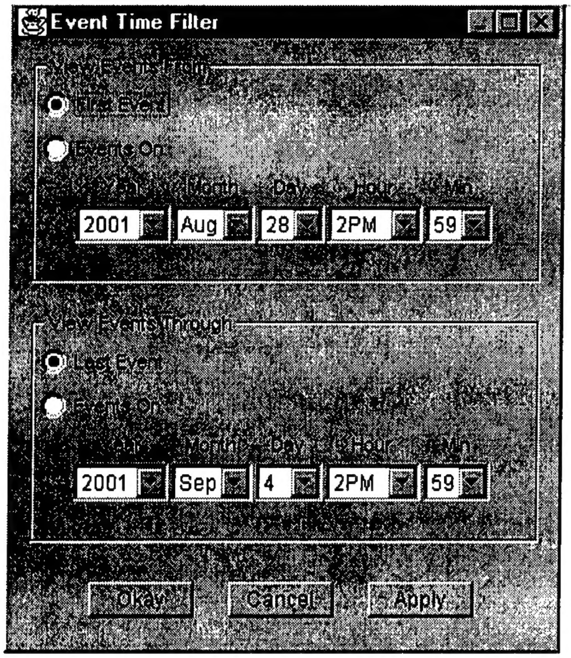



Figure 12: Event log UI

When debugging a composite node, it is often useful to determine how far the messages got in the workflow and examine the contents of the messages, in order to determine whether the workflow has been designed correctly. The Node Contents pane supports this kind of detective work, displaying the nonempty mailboxes and their contents. Individual XML messages can be examined more fully by clicking on the message's table cell (with the red XML text), bringing up a separate pane to display the entire message. The scope of the mailboxes examined can be widened or narrowed as desired, by selecting tree nodes at various levels in the node hierarchy.

The events shown in the Events Pane may be filtered in a number of ways, by adjusting the settings in the filter dialogs accessed through the *View* menu. Events may be filtered with respect to (1) the selected tree node (to view only the events involving the selected node or its descendent nodes); (2) the time the event took place; (3) the type if event; and (4) the type of primitive node involved. The *Event Time Filter* and *Event Type Filter* dialogs are shown in Figure 13. The other filtering options are also accessed via the *view* menu.



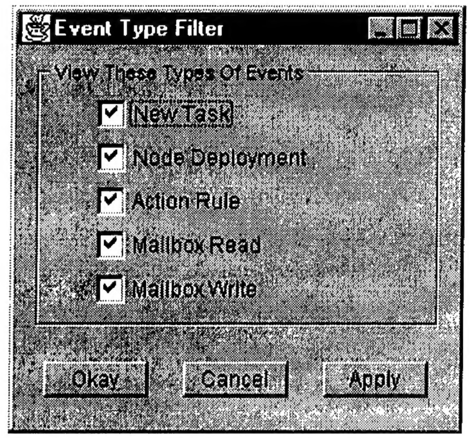



Figure 13: Event filter dialogs

☆U.S. GOVERNMENT PRINTING OFFICE: 2002-710-038-10231

***MISSION***
***OF***
***AFRL/INFORMATION DIRECTORATE (IF)***

*The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.*